

# CSN08x14 Lab 09

## Part A- Python Networking

In part A, this lab gives practical examples for basic networking with Python and for analysing network traffic captures.

All files you need for Part A of this lab can be downloaded from moodle as a single zip archive, labfiles\_Networking.zip.

In part B, we explore encoding of text (strings) to binary, and decoding from binary to text. The script for hashing file contents will be useful for the coursework. Finally, we introduce file type analysis. Again, this is useful for the coursework.

As we are coming closer to the end of the module, some of this lab is more exploratory than earlier ones – the emphasis is on understanding given code examples and then applying concepts from these examples yourself to different situations. This may require you to research selected concepts yourself.

### 1 Sockets

In the lecture, we introduced sockets and discussed how client and server sockets interact. Review the lecture slides on sockets.

#### 1.1 Simple Server using Sockets - Receiving data

The server (recipient) creates a socket then waits for a client to connect and send it some data. Some real-world examples of servers are ftp and telnet which process commands from a client and return the results. We'll create a simple server which receives data from the client, prints it to the screen and returns a confirmation to the client e.g. 'Got that thanks!'.

The script for the server socket is:

```
# server.py
# script to create a server socket to listen for messages
# Owen Lo Nov 2016
# Change Log:
# Oct 2017  Gaye Cleary Updated to python3 & expanded

# For documentation on socket library, see....
# https://docs.python.org/3/library/socket.html
import socket

TCP_PORT = 5005    # The port to use.
BUFFER_SIZE = 1024    # Packet size - 1024 is standard.

def server_socket(tcp_ip, tcp_port):
    # Create a socket object
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # bind the socket to the tcp address and port and start to listen
    sock.bind((tcp_ip, tcp_port))
    sock.listen()
    print(f'#{<INFO> Server Initialising on {tcp_ip}:{tcp_port}')
```

```

print('#<INFO> Awaiting connection...')

# wait for a client to connect
conn, addr = sock.accept()
print(f'#<INFO> A client on: {addr} has connected.')

# process all incoming data from the client....
while True:
    data = conn.recv(BUFFER_SIZE)
    if data:
        print(f'#<INFO> Received: {data.decode("utf-8")}')
        # send a confirmation to the client
        conn.sendall("Got that thanks!".encode('utf-8'))
    else:
        print('#<INFO> End of Data - Socket Exiting...')
        break

conn.close()
sock.close()

def main():
    tcp_ip = socket.gethostbyname('localhost')
    server_socket(tcp_ip, TCP_PORT)

if __name__ == '__main__':
    main()

```

Create an exact copy of the script, or download it from moodle.

Q: what does 'localhost' represent? Why do we use that for the initial test?

Q: what ip address does localhost resolve to?

Q: When creating the socket object, what do the following terms mean?

socket.AF\_INET

socket.SOCK\_STREAM

*Note:* the following webpages are helpful for this question...

<http://whatismyipaddress.com/localhost>

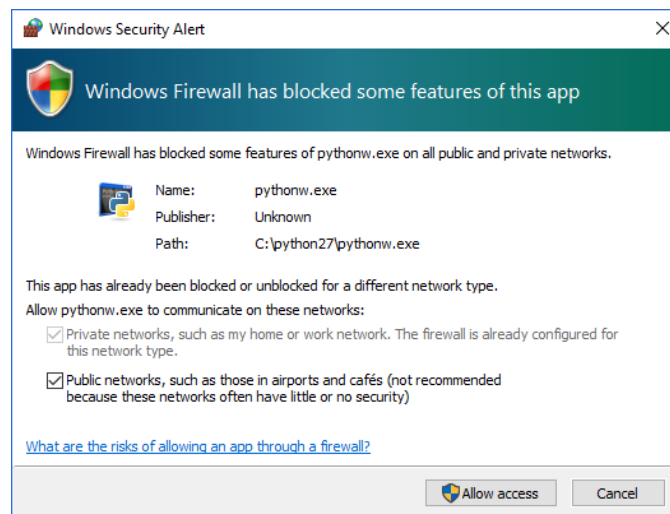
<https://pymotw.com/3/socket/addressing.html>

Go through the script - use the lecture slides to help you understand how the script works, and remind yourself of all the steps required to create the listening socket.

### 1.1.1 Starting your server

Test from the IDLE shell, by running the module using **Run>Run Module**.

You may get a firewall security alert pop up:



You should allow access in this case.

The script should run and indicate where the socket is listening. It will then wait. The output should look like.....

```
>>>
RESTART: C:\Users\CSN08114 Scripting for Cybe
rsecurity and Networks\networking lab\server.py
#<INFO> Server Initialising on 127.0.0.1:5005
#<INFO> Awaiting connection...
```

Q: What is the ip address and port that the server is listening on?

At the windows command prompt, run '**netstat -ano | findstr 5005**' to find details of your server process...

Q: What is the State and process ID of your server?

The server should be in a listening state and the server process id is in the final column.

```
C:\WINDOWS\system32>netstat -ano | findstr /C:"5005 "
```

TCP	127.0.0.1:5005	0.0.0.0:0	LISTENING	8556
-----	----------------	-----------	-----------	------

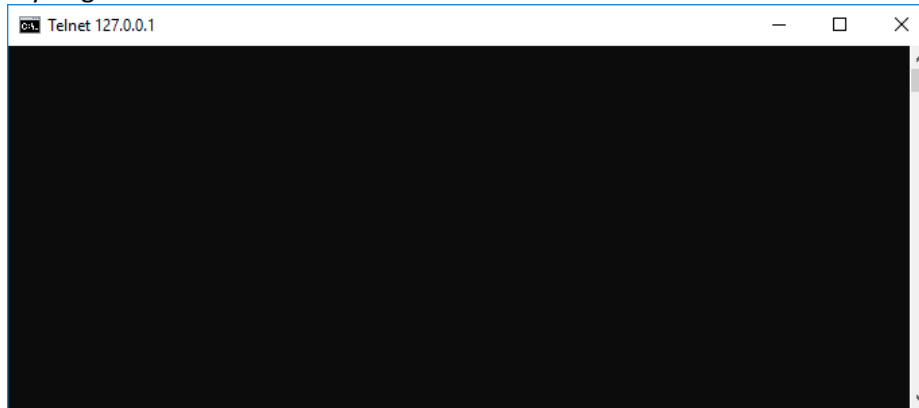
Note: If you need to stop the server running, restart the IDLE shell (Shell > Restart).

### 1.1.2 Client connects to the server – test using windows command line

Open a Windows command prompt.

Type **telnet 127.0.0.1 5005** at the command prompt.

If you get a blank window like this:



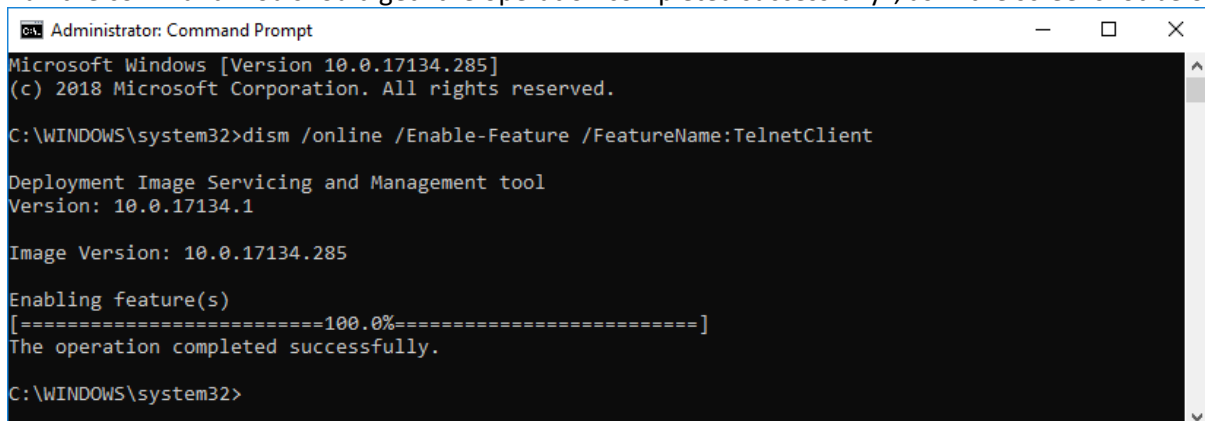
Everything is working and you can continue below (look for the **\*\*\*\*\***).

If you get the error message “'telnet' is not recognized as an internal or external command, operable program or batch file.”, the telnet command is not enabled on your machine.

To enable it, open a second command prompt **in administrator mode**.

Type in **dism /online /Enable-Feature /FeatureName:TelnetClient**

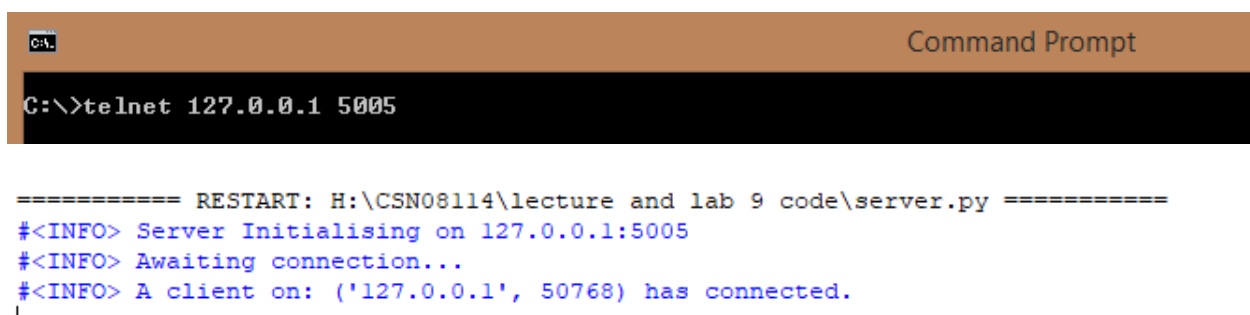
Run the command. You should get “the operation completed successfully”, as in the screenshot below.



If this doesn't work, and/or you don't have admin privileges, move on to Section 1.2.

If it did work, continue at **\*\*\*\*\***.

**\*\*\*\*\*** You can now proceed and connect to your server using the telnet client giving the ip address and port number of your server as parameters. The server will show that a client has connected.



Check the processes that are using port 5005 again e.g. ...

```
C:\WINDOWS\system32>netstat -ano | findstr /C:"5005 "
```

TCP	127.0.0.1:5005	0.0.0.0:0	LISTENING	8556
TCP	127.0.0.1:5005	127.0.0.1:51726	ESTABLISHED	8556
TCP	127.0.0.1:51726	127.0.0.1:5005	ESTABLISHED	9284

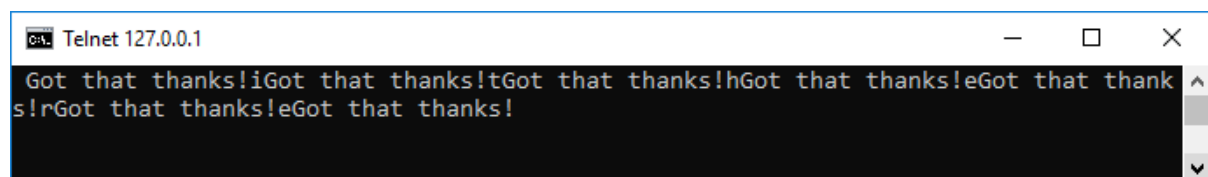
Q: What is the State and process ID of your server?

Q: What is the State and process ID of the telnet client? What port is it running on?

Q: Does the client port reported by the server match the client port for the process?

### 1.1.3 Client sends some data to the server

In the telnet client, type in some text. The telnet and server windows should look something like...



```
===== RESTART: H:\CSN08114\lecture and lab 9 code\server.py =====
#<INFO> Server Initialising on 127.0.0.1:5005
#<INFO> Awaiting connection...
#<INFO> A client on: ('127.0.0.1', 51726) has connected.
#<INFO> Received: h
#<INFO> Received: i
#<INFO> Received: t
#<INFO> Received: h
#<INFO> Received: e
#<INFO> Received: r
#<INFO> Received: e
```

Telnet is sending each letter typed as an individual message; the server confirms each one.

### 1.1.4 Client terminates

Kill the telnet client using the 'taskkill' command as shown in the screenshot below. The PID is the process id of the client found previously, using netstat. Note: take care to kill the client process and not the server.



The server output will look similar to this

```
#<INFO> Received:
Traceback (most recent call last):
  File "C:\Users\██████████\CSN08114 Scripting for Cybersecurity and Networks\networking lab\server.py", line 48, in <module>
    main()
  File "C:\Users\██████████\CSN08114 Scripting for Cybersecurity and Networks\networking lab\server.py", line 43, in main
    server_socket(tcp_ip, TCP_PORT)
  File "C:\Users\██████████\CSN08114 Scripting for Cybersecurity and Networks\networking lab\server.py", line 29, in server_socket
    data = conn.recv(BUFFER_SIZE)
ConnectionResetError: [WinError 10054] An existing connection was forcibly closed by the remote host
>>> |
```

### 1.1.5 \*\*OPTIONAL\*\* Error handling when the client closes the connection

Add 'try' and 'except' statements around the 'receive' / 'sendall' block to trap the socket shutdown and exit cleanly, printing an informational message, similar to the one below.

```
>>>
RESTART: C:\██████████\CSN08114 Scripting for Cybersecurity and Networks\networking lab\server.py
#<INFO> Server Initialising on 127.0.0.1:5005
#<INFO> Awaiting connection...
#<INFO> A client on: ('127.0.0.1', 54400) has connected.
#<INFO> Socket Shutdown Received - Exiting...
>>> |
```

### 1.1.6 Simple Server using Sockets – recap

Q. Explain, in your own words, how the script works. (Hint: check the steps for creating a server socket from the lecture and match each of these steps to a line or block of the code)

## 1.2 Client & Server using sockets

### 1.2.1 Creating a Client

Now create a client script. A starter script is shown below or it can be downloaded from moodle.

```
# client.py
# start script to create send message to a listening server socket
# Owen Lo Nov 2016
# Change Log:
# Oct 2017 Gaye Cleary Updated for 3.6 & expanded

# For documentation on socket library, see....
# https://docs.python.org/3/library/socket.html
import socket
import sys

BUFFER_SIZE = 1024 #Packet size.

def client_socket(tcp_ip, tcp_port, message):

    # enter your code here to....
    # 1) create a local socket and
    # 2) connect it to the server socket
    pass
    pass
    print(f'##<INFO> Connected to server {tcp_ip}, {tcp_port}')

    # enter your code here to....
    # 3) send the message to the server and
    # 4) receive and decode the server's reply
    pass
    decoded_data = ''
    print(f'##<INFO> Reply Received:', decoded_data)

    # enter your code here to....
    # 5) close the socket
    pass

def main():

    message = input ('Enter a message to send to the server: ')

    tcp_ip = '127.0.0.1'
    tcp_port = '5005'
    client_socket(tcp_ip, int(tcp_port), message)

    print(f'##<INFO> exiting...')

# Standard boilerplate code to call the main() function to begin
# the program if run as a script.
if __name__ == '__main__':
    main()
```

Test the starting script by running it in IDLE. It should run, but not do much yet! Now complete the script.

Tip: Use the server script and lecture slides for information and example code.

Once you think you have the script completed, first Check the module (Run > Check module) for syntax errors. If that works ok, you are ready to see if you can send a message!

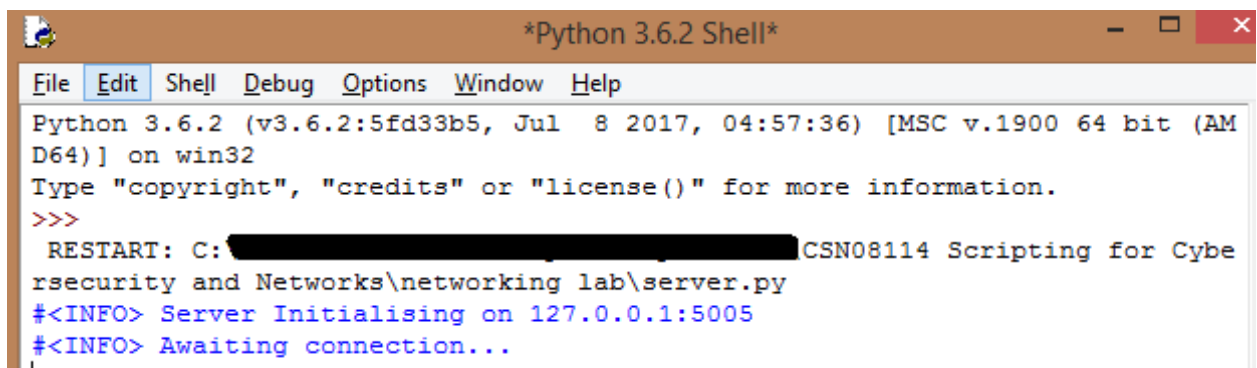
Q: In which order do we need to run the two scripts? Why?

We need to run server.py first, so that it can start listening and be waiting when the client runs.

**Then open a second IDLE shell** and run client.py in this second shell.

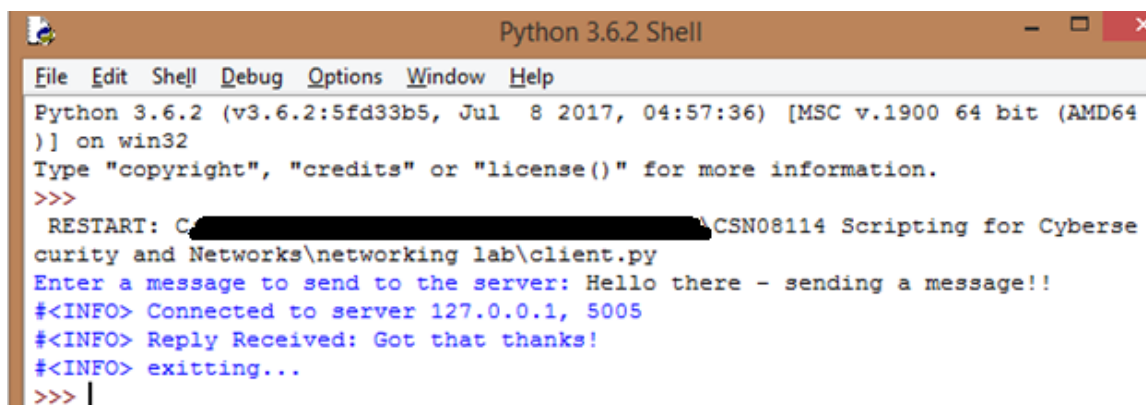
You should be able to send a message from the client and see this has been received in the IDLE output for the server. The output will look similar to this...

Server started and listening:



```
*Python 3.6.2 Shell*
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\[redacted]\CSN08114 Scripting for Cybersecurity and Networks\networking lab\server.py
#<INFO> Server Initialising on 127.0.0.1:5005
#<INFO> Awaiting connection...
|
```

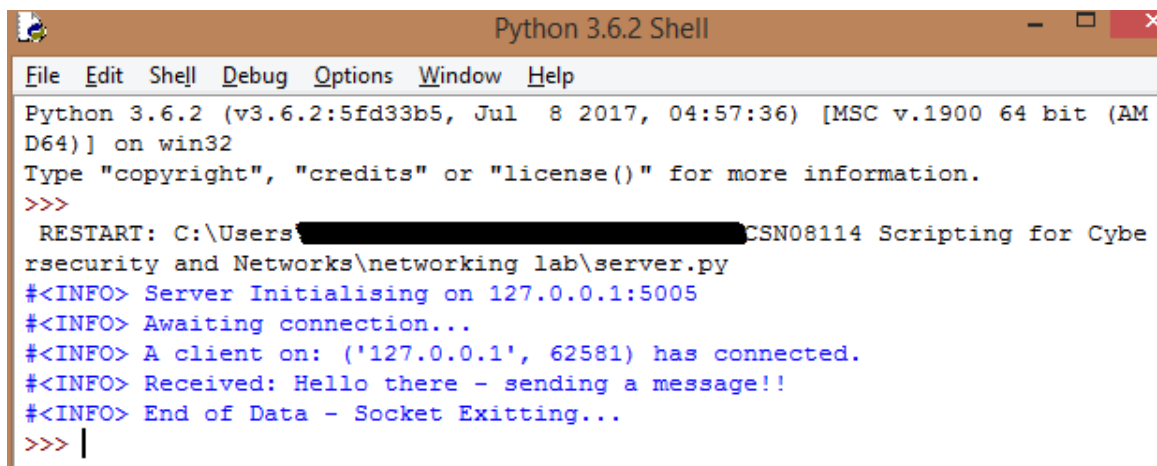
Client started, message entered, transmitted and confirmation received:



```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\[redacted]\CSN08114 Scripting for Cybersecurity and Networks\networking lab\client.py
Enter a message to send to the server: Hello there - sending a message!!
#<INFO> Connected to server 127.0.0.1, 5005
#<INFO> Reply Received: Got that thanks!
#<INFO> exiting...
>>> |
```



Server receives message and exits cleanly:



```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\Users\CSN08114 Scripting for Cybersecurity and Networks\networking lab\server.py
#<INFO> Server Initialising on 127.0.0.1:5005
#<INFO> Awaiting connection...
#<INFO> A client on: ('127.0.0.1', 62581) has connected.
#<INFO> Received: Hello there - sending a message!!
#<INFO> End of Data - Socket Exiting...
>>> |
```

Notes:

If something unexpected happens (e.g. cannot receive message from client or the server will not start due to exceptions) make sure the server is not already running in the background by running netstat at the Windows Command Prompt, i.e.: `cmd > netstat -ano | findstr 5005`

If that port (5005) is in use, kill it using the taskkill command used previously. Then run server.py again. Alternatively, use another port if 5005 is taken.

Q: Why do we need to use two separate IDLE shells?

Q: Why do we use the server's name or address in both scripts? Why don't we need to know the client's address?

### 1.2.2 **\*\*OPTIONAL\*\*** Understanding BUFFER\_SIZE

To verify that you really understand the purpose of the BUFFER\_SIZE variable and the loop in the server script, change the BUFFER\_SIZE in server.py to 10 and test the scripts with the message...  
my message is longer than the buffer

Q: What happens to the message received by the server ?

Q: Why do we use a while loop and not a for loop? How exactly does this while loop work?

Note: Before continuing, change the buffer size back to 1024.

### 1.2.3 **\*\*OPTIONAL\*\*** Networked Client / Server

(This may not work in the lab due to security settings)

So far, both the client and server have been running on the local machine using '127.0.0.1' as the ip address. A more flexible solution would be to have the client and server run on different machines communicating over a network. To achieve this, follow these steps...

Step 1: Amend your server to identify the hostname using `socket.gethostname()` and use that result to obtain the ip address. i.e.

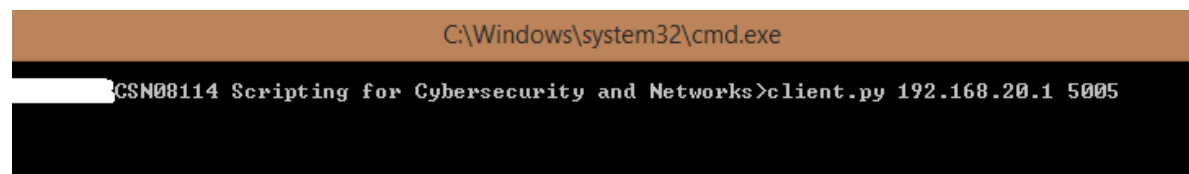
```
tcp_ip = socket.gethostbyname(socket.gethostname())
```

Run the server and examine the output

Q: What ip address is your server running on?

Q: What is your machine's IP address? (Hint: run *ipconfig* from windows command line)

Step 2: Amend the client script to accept the ip address and port as command-line arguments e.g



The screenshot shows a Windows command prompt window with the title bar 'C:\Windows\system32\cmd.exe'. The command prompt shows the path 'C:\Users\CSN08114\Scripts' and the command 'client.py 192.168.20.1 5005' being executed. The output is not visible in the screenshot.

Run the client supplying the ip address and port of the server as parameters.

Once this works, pair up with a friend and try sending a message from one lab machine to another – one of you will be the server, the other the client.

## 2 Analysing Network Traffic

### 2.1 Working with pcap packet captures – using the dpkt module

Check that dpkt is already installed on your machine:

```
>>> help('modules')
```

The script **parse\_pcap.py**, shown on the next page, extracts and parses the records in a packet capture (pcap) file.

Create a copy of this script (or you can download it from moodle).

Also download the sample packet captures **filtered2.pcap** and **filtered3.pcap** from moodle.

```

# parse_pcap.py
# script to parse a pcap file
# Adapted from:
#   https://jon.oberheide.org/blog/2008/10/15/dpkt-tutorial-2-parsing-a-pcap-
#   file/)
# Oct 2017  Gaye Cleary Updated for 3.6 & added functionality

# For documentation on dpkt library, see....
# https://dpkt.readthedocs.io/en/latest/

import dpkt

def main():
    pcapfile = 'filtered2.pcap'
    f = open(pcapfile, 'rb')
    pcap = dpkt.pcap.Reader(f)

    for ts, buf in pcap:
        eth = dpkt.ethernet.Ethernet(buf)
        print(f'#{<INFO> ethernet packet: {repr(eth)}')

        ip = eth.data
        print(f'#{<INFO> ethernet packet: {repr(ip)}')
        # tcp = ip.data
        # print(f'#{<INFO> ethernet packet: {repr(tcp)}')

        # print(f'{ip.src}:{tcp.sport} -> {ip.dst}:{tcp.dport}')

        break

    f.close()

# Standard boilerplate code to call the main() function to begin
# the program if run as a script.
if __name__ == '__main__':
    main()

```

### 2.1.1 Familiarisation with pcap – using Wireshark

Network communications are achieved by a process called encapsulation. To see how this looks, open **filtered2.pcap** using wireshark and look at the first packet. It will look like this (some interesting fields have been circled)...



### 2.1.2 Familiarisation with pcap records – using dpkt

parse\_pcap.py just prints out the first ethernet frame. Examine the output and try to understand how it's formatted. The output should be similar to that shown below

```
...rity and Networks\networking lab\parse_pcap.py
#<INFO> ethernet packet: Ethernet(dst=b'\xa4\x18uGID', src=b'\xe8@\xf2;\xeb\xcd'
, data=IP(len=1010, id=958, off=16384, ttl=128, p=6, src=b'\x92\xb0\xa4[' , dst=b
'\x17\x156\x83', opts=b'', data=TCP(sport=53954, dport=80, seq=846336224, ack=54
3676321, flags=24, win=64647, sum=34952, opts=b'', data=b'GET /ping?h=foxnews.co
m&p=%2Ftech%2F2016%2F11%2F04%2Fturkey-blocks-popular-social-networks.html&u=CLcz
ulowhFLB2D0zk&d=foxnews.com&g=8971&g0=tech&g1=Fox%20News&n=1&f=00001&c=0.25&x=0&
m=0&y=4413&o=1903&w=935&j=30&R=1&W=0&I=0&E=10&e=10&v=%2Ftech.html&K=490::1025::D
Q3PPhDzKEf4BfkV0pDzxtC-DUi0Jg:.*%5B%40id%3D%27content%27%5D%2Fdiv%5B%5D%2Fsecti
on%5B%5D%2Ful%5B%5D%2Fli%5B%5D%2Farticle%5B%5D%2Fdiv%5B%5D%2Fa%5B%5D::c::h
ttp%3A%2F%2Fwww.foxnews.com%2Ftech%2F2016%2F11%2F04%2Fturkey-blocks-popular-soci
al-networks.html::mYfaFDaxKAmCUEVsvD0Q03JCF20-&b=441&t=CVtDhZLp58j6oP4SDuFeG-DM
os5e&V=85&tz=0&sn=2&eM=4021&EE=10&sv=D0-nWBDQwxyOBgd0oEDfCj00CXr2V4&_ HTTP/1.1\r
\nHost: ping.chartbeat.net\r\nUser-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv
:47.0) Gecko/20100101 Firefox/47.0\r\nAccept: */*\r\nAccept-Language: en-GB,en;q
=0.5\r\nAccept-Encoding: gzip, deflate\r\nReferer: http://www.foxnews.com/tech/2
016/11/04/turkey-blocks-popular-social-networks.html\r\nConnection: keep-alive\r
\n\r\n'))
>>> |
```

Q: what type of object is eth?

Q: can you identify the src and dst IP addresses?

Q: can you identify the src and dst TCP ports?

Q: what sort of object is the TCP.data field?

eth is an instance of the Ethernet class. There are three attributes with values, dst, src and data. The data element is an instance of the IP class with many fields including src and dst, and another data field. This IP.data is an instance of the TCP class – again containing many fields including sport and dport, and a final data field – the HTTP packet as a binary stream. More details of these classes can be found using built-in help e.g. help(dpkt.ethernet.Ethernet)

```
>>> help(dpkt.ethernet.Ethernet)
Help on class Ethernet in module dpkt.ethernet:

class Ethernet(dpkt.dpkt.Packet)
|   Ethernet.
|
|   Ethernet II, LLC (802.3+802.2), LLC/SNAP, and Novell raw 802.3,
|   with automatic 802.1q, MPLS, PPPoE, and Cisco ISL decapsulation.
|
|   Attributes:
|       __hdr__: Header fields of Ethernet.
|       TODO.
|
```

### 2.1.3 Parsing pcap records using dpkt

Expand *parse\_pcap.py* to create new variables for the ip and tcp objects, then print the connection details (source ip addr and port & destination ip addr and port) for each pcap record. Note: remember to remove the *'break'* statement which exits the loop after the 1st record is read. The source code and output might look like this.

```
ip = eth.data
tcp = ip.data
print(f'Connection details: {ip.src}:{tcp.sport} -> {ip.dst}:{tcp.dport}')
```

```
Connection details: b'\x92\xb0\xa4[':54261 -> b'\x17\xd1\xd2\xf2':80
Connection details: b'\x92\xb0\xa4[':54333 -> b'\xd4:\xf6m':80
Connection details: b'\x92\xb0\xa4[':54289 -> b'\x17\xd1\xd2\xf2':80
Connection details: b'\x92\xb0\xa4[':54304 -> b'\x17\xd1\xd2\xf2':80
Connection details: b'\x92\xb0\xa4[':54289 -> b'\x17\xd1\xd2\xf2':80
Connection details: b'\x92\xb0\xa4[':54304 -> b'\x17\xd1\xd2\xf2':80
Connection details: b'\x92\xb0\xa4[':54289 -> b'\x17\xd1\xd2\xf2':80
Connection details: b'\x92\xb0\xa4[':54289 -> b'\x17\xd1\xd2\xf2':80
Connection details: b'\x92\xb0\xa4[':54327 -> b'6\xcc\x06\xe':80
>>> |
```

Q: What data type are the ip addresses?

This output is correct, but the ip addresses are not in the format we're used to!

### 2.1.4 Reformat IP address

ip addresses are generally shown in this format: '127.0.0.1'. But in the pcap record they're a 32 bit binary streams e.g. *src=b'\x92\xb0\xa4['*, *dst=b'\x17\x156\x83'*. Each byte (8 bits) represents one of the octets in the dotted decimal format.

e.g *b'\x92\xb0\xa4['* is 146.176.164.91

because: *\x92* = 146 decimal; *\xb0* = 176 decimal; *\xa4* = 164 decimal and *[* is 91 decimal.

The *socket.inet\_ntoa()* function, part of the standard library module *socket*, decodes IP addresses from the binary format stored in network captures to the format we are used to.

Add the import statement and *decode\_ip* function shown below to your code:

```
import socket

def decode_ip(binary_ip):
    """decodes ip address from binary to standard octets"""
    return socket.inet_ntoa(binary_ip)
```

and replace the print statement with

```
print(f'{decode_ip(ip.src)}:{tcp.sport} -> {decode_ip(ip.dst)}:{tcp.dport}')
```

Test your script. Your output should now look like this (the screenshot shows the first few lines):

```
146.176.164.91:53954 -> 23.21.54.131:80
146.176.164.91:54040 -> 66.117.29.34:80
146.176.164.91:54040 -> 66.117.29.34:80
146.176.164.91:53954 -> 23.21.54.131:80
146.176.164.91:53630 -> 72.247.177.201:80
```

See the optional extension exercise in moodle if you'd rather decode the octets from first principles, using bit shifts.

### 2.1.5 Extract http Requests from the pcap file

Next, we'll drill down a bit further into the http packets.

Read the following short tutorial on dpkt at <https://jon.oberheide.org/blog/2008/10/15/dpkt-tutorial-2-parsing-a-pcap-file/> then use `dpkt.http.Request()` to extract http requests sent to a web server (i.e. records with a destination port of 80).

To familiarise yourself with the format of the http records, print it using `repr()`. This will show the different attributes in the record, and their values. The output should look similar to that shown below – the 1<sup>st</sup> 2 attributes have been highlighted in red.

Note: if you get errors when decoding the http traffic, add some error handling using 'try' and 'except' statements to trap the problem records – these can be ignored right now.

```
.....rity and Networks\networking lab\parse_pcap.py
#<INFO> HTTP request: Request(version='1.1', method='GET',
uri='/ping?h=foxnews.com&p=%2Ftech%2F2016%2F11%2F04%2Fturkey-blocks-popular-
social-
networks.html&u=CLczulowhFLB2D0zk&d=foxnews.com&g=8971&g0=tech&g1=Fox%20News&n
=1&f=00001&c=0.25&x=0&m=0&y=4413&o=1903&w=935&j=30&R=1&W=0&I=0&E=10&e=10&v=%2F
tech.html&K=490::1025::DQ3PPPhDzKEf4BfkV0pDzxtC-
DUi0Jg::*%5B%40id%3D%27content%27%5D%2Fdiv%5B3%5D%2Fsection%5B1%5D%2Ful%5B1%5D
%2Fli%5B1%5D%2Farticle%5B1%5D%2Fdiv%5B1%5D%2Fa%5B1%5D::c::http%3A%2F%2Fwww.fox
news.com%2Ftech%2F2016%2F11%2F04%2Fturkey-blocks-popular-social-
networks.html::mYfaFDAXKAmCUEVsvD0QO3JCFE2O-&b=441&t=CVtDhZLp58j6oP4SDuFeG-
DMos5e&V=85&tz=0&sn=2&em=4021&EE=10&sv=D0-nWBDQwxyOBgd0oEDfCj00CXr2V4&_',
headers=OrderedDict([('host', 'ping.chartbeat.net'), ('user-agent',
'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0'),
('accept', '*/*'), ('accept-language', 'en-GB,en;q=0.5'), ('accept-encoding',
'gzip, deflate'), ('referer', 'http://www.foxnews.com/tech/2016/11/04/turkey-
blocks-popular-social-networks.html'), ('connection', 'keep-alive'))],
body=b'', data=b'')
>>>
```

Q: What are the other two attributes in a http record?

Q: How would you extract the 'method' values from the http record?

Q: How would you extract the 'referer' values from http.headers?

The following lines of code will print the full http record.

```
if tcp.dport == 80:
    http = dpkt.http.Request(tcp.data)
    print(f'#{<INFO> HTTP request: {repr(http)}')
```

The http record has four attributes: version, method, uri and headers. These 4 attributes are referenced as `http.version`, `http.method`, `http.uri` and `http.headers`. However, headers is a `OrderedDict` object (for more details, see <https://docs.python.org/3/library/collections.html#collections.OrderedDict>). One of its keys is 'referrer', therefore the associated values can be found using `http.headers['referrer']`

Finally, update your script to print the 'method' and 'referrer' for each http record.

### 2.1.6 Extract a gif

The script `pcap_downloads.py`, shown below, extracts the downloaded gif images from a packet capture.

```
# pcap_downloads.py
# pcap packet analysis script to find downloaded gifs
# possible solution for lab exercise
# adapted from: Violent Python Ch 4 (p136)
# Petra Leimich Nov 2016
# Change Log:
#   Oct 17: Gaye Cleary - updated to pthon 3.6
#   Nov 18: PL - Python 3.7 / PEP8
import dpkt
import socket

def findDownload(pcap):
    '''in current form, finds any gif files downloaded and prints
       request source (Downloader), gif URI and destination (provider) IP'''

    found = False
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            dst = socket.inet_ntoa(ip.dst)
            tcp = ip.data

            http = dpkt.http.Request(tcp.data)
            if http.method == 'GET':
                uri = http.uri.lower()
                if '.gif' in uri:
                    print(f'[{!}] {src} downloaded {uri} from {dst}')
                    found = True
        except Exception:
            # necessary as many packets would otherwise generate an error
            pass
    return found

def main():
```



```

# should get results with filtered2.pcap but none with filtered3.pcap
pcapFile = 'filtered2.pcap'
f = open(pcapFile, 'rb')
pcap = dpkt.pcap.Reader(f)

print(f'[*] Analysing {pcapFile} for gif files')
# call findDownload which prints results
result = findDownload(pcap)
if result is False:
    print('No gif downloads found in this file')

if __name__ == '__main__':
    main()

```

Run the script twice, first with the sample filtered2.pcap, then with filtered3.pcap. You should get the following output:

```

>>>
RESTART: C:\[redacted]CSN08114 Scripting for Cybersecurity and
Networks/networking lab/pcap_downloads.py
f[*] Analysing {pcapFile} for gif files
[!] 146.176.164.91 downloaded /static/all/img/clear.gif from 23.209.213.8
[!] 146.176.164.91 downloaded /o.gif?~rs~s~rs~news~rs~t~rs~highweb_story~rs~i~rs~3787289
9~rs~p~rs~99207~rs~a~rs~domestic~rs~u~rs~/news/uk-politics-37872899~rs~r~rs~0~rs~q~rs~0~
rs~z~rs~981~rs~ from 212.58.244.39
>>>
RESTART: C:\[redacted]SN08114 Scripting for Cybersecurity and
Networks/networking lab/pcap_downloads.py
f[*] Analysing {pcapFile} for gif files
No gif downloads found in this file
>>>

```

Your task is to figure out how exactly the script works! You will need to do some research to do this, the lecture itself won't be enough.

If needed, refer again to the short dpkt tutorial at <https://jon.oberheide.org/blog/2008/10/15/dpkt-tutorial-2-parsing-a-pcap-file/>

Q: In Wireshark, we used the filter  
**http.request.method==GET && http.request.full\_uri matches "http://.\*\gif.\*"**  
 What are the equivalents of these two conditions used in Python?

Now that you have (hopefully) understood the script, adapt it to find downloaded jpg files instead. Test the adapted script with the two packet captures.

### 2.1.7 **\*\*Optional\*\*** Challenge Question – Analyse network flows

**Note:** This question will require you to investigate methods for sorting a dictionary on the value fields.

To get a quick view of the network traffic in the pcap file, let's have a look at the network flows – i.e. the number of packets flowing between each source ip address and port to each destination ip address and port.

One way to achieve this is to create a dictionary where the keys are a concatenation of the source (ip address and port) and destination (ip address and port). For each pcap record, if the key is already in the dictionary, increment its value. If not, add an entry for this key, with a value of 1.

Once the dictionary is populated, it can be sorted into a list (sorting on the 'value' field) and the top flows of traffic printed. Sorting a dictionary is tricky but there are quite a few ways to go about it. Using the `itemgetter()` function (in the `operator` module) is one option (see <https://docs.python.org/3/howto/sorting.html>). Another is to use a lambda function to set the key (see [https://www.youtube.com/watch?v=MGD\\_b2w\\_GU4](https://www.youtube.com/watch?v=MGD_b2w_GU4))

The output will look something like this....

```
>>> #<INFO> print top 10
146.176.164.91:53659->23.209.213.8:80 - 13
146.176.164.91:54261->23.209.210.242:80 - 11
146.176.164.91:53656->23.209.213.8:80 - 10
146.176.164.91:53639->72.247.177.201:80 - 8
146.176.164.91:53862->23.209.213.8:80 - 8
146.176.164.91:53655->23.209.213.8:80 - 8
146.176.164.91:54260->212.58.244.68:80 - 7
146.176.164.91:54264->23.209.210.242:80 - 7
146.176.164.91:54289->23.209.210.242:80 - 7
146.176.164.91:53664->2.22.228.18:80 - 6
>>>
```

### 2.1.8 **\*\*Optional\*\*** Challenge Question – detect packets from an unknown source

Use methods from the previous scripts and make use of your understanding of the `dpkt` module to write a script that will list all packets in a packet capture that are **not** from a specified source.

As our two sample captures were created from a PC with the address 146.176.164.91, most of the packets will show this address as the source.

Therefore, your script, `pcap_exclude.py`, should show all packets from a capture that did NOT come from this source. The output should be formatted as follows...

```
[+] Src: 212.58.244.68 --> Dst: 146.176.164.91
```

Test your script with `filtered2.pcap`. Your output should look like this:

```
p_analysis.py
[*] analysing filtered2.pcap for packets not source 146.176.164.91
-----
[+] Src: 54.194.240.68 --> Dst: 146.176.164.91
[+] Src: 54.69.185.4 --> Dst: 146.176.164.91
[+] Src: 54.69.185.4 --> Dst: 146.176.164.91
[+] Src: 204.2.197.201 --> Dst: 146.176.164.91
[+] Src: 54.69.185.4 --> Dst: 146.176.164.91
[+] Src: 151.101.16.233 --> Dst: 146.176.164.91
[+] Src: 52.1.64.28 --> Dst: 146.176.164.91
[+] Src: 54.210.45.182 --> Dst: 146.176.164.91
[+] Src: 54.86.76.22 --> Dst: 146.176.164.91
[+] Src: 54.163.85.105 --> Dst: 146.176.164.91
[+] Src: 212.58.244.68 --> Dst: 146.176.164.91
[+] Src: 77.72.112.213 --> Dst: 146.176.164.91
[+] Src: 212.58.244.68 --> Dst: 146.176.164.91
```

Hint:

Create your script a bit at a time, using the previous scripts for inspiration.

Which variable contains the source IP address? – you will need an if statement that checks the value of this variable for each packet and prints output only if the value is not the specified address that is to be excluded.

## Part B: bytes, strings, hashing

### 3 Working with Bytes and Strings - encoding and decoding

When any string is stored on a computer, it needs to be **encoded** into binary, or bytes. As we saw in the lecture, there are various methods of doing this, such as ascii and UTF-8.

When a text file is read from storage for human consumption, the bytes need to be **decoded** back into human readable strings. Python 3 distinguishes clearly between bytes and strings - for example, to hash a file using hashlib requires bytes.

It is important to decode using the same codec as was used originally for encoding. We saw in the lecture that the first two bytes of text files may give us a clue as to what codec was used. However, in general it is a very complex task to try and determine the encoding used just from the byte stream stored in the file. There are some Python libraries to help, but this is way beyond the contents of this module.

So for now, let's do a simple exercise to demonstrate why it matters.

In IDLE, execute the commands below.

Note: To type e.g. French accents and German umlauts on a UK keyboard, you have various options. You could use the Alt key with a number combination. To do this, hold down the Alt key, then press the required number combination. This must be on the numeric keypad - the numbers at the top of the main keyboard won't work! For example, é is Alt+130. Try this in IDLE - it works for me. (see e.g. <http://www.frenchpropertylinks.com/frenchcharactersenglishukkeyboard.htm> for a list)

```
>>> import os
>>> os.getcwd()
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
```

Q: What does the last line of code do?

Q: You will notice that the final command returns the value 4. Why is it 4?

Now try

```
>>> open('cafe.txt').read()  
>>> open('cafe.txt', encoding='utf_8').read()
```

Q: Which one of these commands returns the actual text you wrote to the file?

Q: Why does the other command return 'cafÃ©'?

## 4 File content hashing

Previously we looked at Hash Signatures of short strings such as passwords. In digital forensics, Hash Signatures of entire files are commonly used to match evidence files to known files.

Note: Even if file names or extensions have been changed, the hash of the contents will still match so we can find contraband files this way!

The following exercises will take you through file hashing.

Note that all parts below feed into the coursework for the module

## 4.1 Create a Hash Signature for a File

Create a file `C:\temp\a_file.txt`, and add some text.

Now use Python to create an MD5 signature for the file. This is the **hash of the contents of the file**. Use code similar to the following, and remember to import hashlib first!

```
>>> f = open('C:\\temp\\a_file.txt', 'rb')
>>> file_content = f.read()
>>> md5_obj = hashlib.md5(file_content)
>>> print(md5_obj.hexdigest())
8d9a06a52e6d553bedd1d0ba19ff00af
```

Remember to close the file afterwards.

Q: What is the hash signature generated?

Q: What is the length of the hash signature?

Q: Do you think the hash signature would change if we were to change the name of the file?

Let's test this. Change the name of the file and create another MD5 signature for the file.

Q: What is the hash signature generated?

Q: Why hasn't it changed?

The contents are hashed, and the name and other metadata should not make any difference to the signature. This way extension or name changes to files should not stop out forensics tools from finding contraband files!

Q: Compare the `md5()` method call above with how we used it on strings in previous labs. What is different?

Here, we didn't need to specify the encoding to be used during hashing. This is because we opened the file in binary mode, so Python read bytes rather than the strings these bytes represent.

Try the code above using 'r' rather than 'rb' as the mode when opening the file. This should generate the error

```
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    md5_obj = hashlib.md5(file_content)
TypeError: Unicode-objects must be encoded before hashing
```

So, the md5() function must always be called with a byte object, not a string.

If you open a file as binary, its contents are read as a byte object which can be hashed directly, but if you open a file without the 'b' mode, its contents are read as a string which needs to be encoded before you can hash it, using file\_content.encode('utf-8') or similar. We discussed encoding in the lecture.

## 4.2 File Hash Signature Generator Script

Now create a script **file\_hash.py**, with a **get\_hash** function which takes the filename as an argument, based on the code below.

You can copy & paste the code from below to create the initial script, or download it from moodle.

```
# Script: file_hash.py
# Desc:   Generate file hash signature - start code
# modified: 12/11/18 (PEP8 compliance)
#
import sys
import os
import hashlib

def get_hash(filename):
    """prints a hex hash signature of the file passed in as arg"""
    try:
        # Read File
        # *** your code ***

        # Generate Hash Signature
        # *** your code ***

        print(f'[+] get_hash() file: {filename} ', end='')
        # print(f'hash_sig: {file_hashsig}')

    except Exception as err:
        print(f'[-] {err}')

    finally:
        if 'f' in locals():
            f.close()

def main():
    # Test case
    sys.argv.append(r'c:\temp\a_file.txt')
    # Check args
    if len(sys.argv) != 2:
        print(f'[-] usage: file_hash filename')
        sys.exit(1)

    filename = sys.argv[1]
    get_hash(filename)
```

```
if __name__ == '__main__':  
    main()
```

Test from the IDLE shell, by importing the module and calling the function.

```
file_hash.get_hash('<filename>').
```

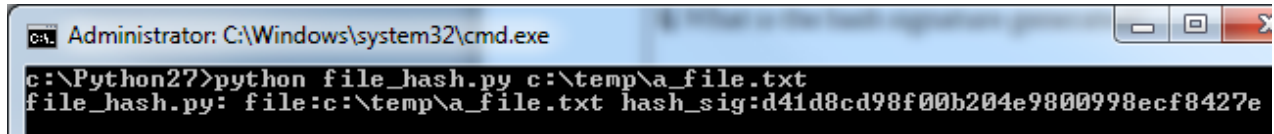
Once you have completed the script, you should get similar output to the following (depending on the file content you hash):

```
>>>  
>>> file_hash.get_hash('c:\\temp\\a_file.txt')  
file_hash.py: file:c:\\temp\\a_file.txt hash_sig:d41d8cd98f00b204e9800998ecf8427e  
>>>
```

Q: What is the hash signature generated?

Q: What does the "finally:" block do? (read <https://docs.python.org/3.6/tutorial/errors.html> to find out!)

Test from the command line:



```
C:\Python27>python file_hash.py c:\temp\a_file.txt  
file_hash.py: file:c:\temp\a_file.txt hash_sig:d41d8cd98f00b204e9800998ecf8427e
```

### 4.3 Create a Lookup Table of Bad File Hashes

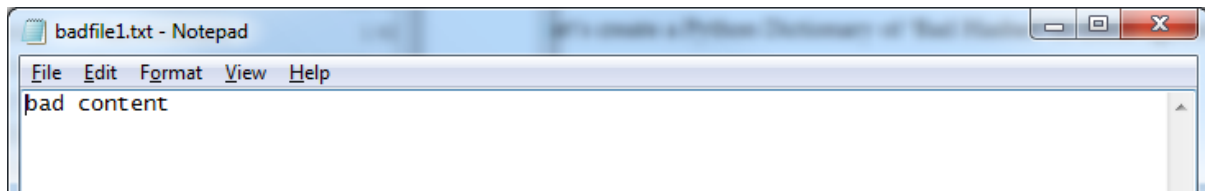
Change the script to **return** the file hash signature (instead of printing it) by replacing the print with a **return** command: (move the print statement to main()) so it still prints if module called as a script)

From IDLE shell test:

```
>>>  
>>> file_hash.get_hash('c:\\temp\\a_file.txt')  
'd41d8cd98f00b204e9800998ecf8427e'  
>>> |
```

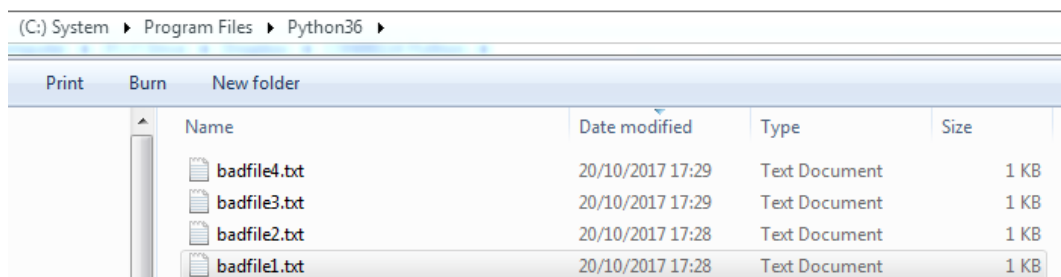
Now let's create some files with 'bad content'.

In the **Python37** directory (C:\Program Files\Python37) create a file '**badfile1.txt**' using Explorer. Edit the file and add the text '**bad contents**' to the file and save.



Using **Save As**, create 4 'bad' files:

- badfile1 with text 'bad content'
- badfile2 with text 'bad stuff'
- badfile3 with text 'bad minton'
- badfile4 with text 'bad teacher'



From the interpreter, let's use our `get_hash()` function to create a Python Dictionary of 'Bad Hashes' from the files, using the hash as the key, and the filename as the value: (use a for and the range() BIF if you want to be really 'Pythonic')

```
>>> import file_hash
>>> bad_files = {}
>>> bad_files[file_hash.get_hash('badfile1.txt')] = 'badfile1.txt'
>>> bad_files[file_hash.get_hash('badfile2.txt')] = 'badfile2.txt'
>>> bad_files[file_hash.get_hash('badfile3.txt')] = 'badfile3.txt'
>>> bad_files[file_hash.get_hash('badfile4.txt')] = 'badfile4.txt'
>>> bad_files.items()
dict_items([('4eea91e7a2009ecd9bbf1242b4eaf267', 'badfile1.txt'), ('cff5ad7d18e86a83e8b0660d5707397a', 'badfile2.txt'), ('4b909a85b6216851eb7197949bbb29b1', 'badfile3.txt'), ('2b5a2771906feb3baf75b35675a8919b', 'badfile4.txt')])
```

Let's test that we can match on the hash signatures. Get the hash of one of the bad files:

```
>>> hash_sig = file_hash.get_hash('badfile4.txt')
>>> print(hash_sig)
2b5a2771906feb3baf75b35675a8919b
```

Check if the bad file hash is in the bad file dictionary, and if so print out the bad file name, with code such as:

```
>>> if hash_sig in bad_files:
    print(f'[+] Bad file match: {bad_files[hash_sig]} hash: {hash_sig}')
```



Q: Is the bad file hash found?

Q: Why is a Python Dictionary a good way of storing the bad file hash details?

#### 4.4 Test on Content of New Files

Create 2 new files in C:\Program Files\Python37:

unsure1.txt with text 'good content'

unsure2.txt with text 'bad stuff'



Check if the bad file hash is in the bad file dictionary, and if so print out the bad file name, adapting the code from the Exercise above.

Q: Is the unsure1.txt file in the bad files lookup table?

Q: Is the unsure2.txt file in the bad files lookup table?

Q: Which bad file does it have the same contents as?

Q: If you add a space to the end of the content of unsure2.txt, is it still identified? Explain.

## 5 **\*\*Optional\*\*** Digital Forensics – File Type Analysis

**This exercise is useful for the coursework.** However, no other parts of the coursework depend on it. So, if you feel that you are a bit overwhelmed, this component can be left out and you can still get good marks for the rest of the coursework.

File signature analysis can be used to determine the type a file is. This is a common forensic analysis process. Typically the first few bytes of a file are read and the hex representation of these bytes is compared against a list of file signatures to find the type of file. Most file types can be identified by their hex signature (also called "magic number"), and lists of such signatures are built into most forensic tools. In linux, the "file" command determines the type of a file from its signature.



Details of common file type signatures can be found at:  
[http://en.wikipedia.org/wiki/List\\_of\\_file\\_signatures](http://en.wikipedia.org/wiki/List_of_file_signatures)

A common method of hiding bad files is to change the file extension.

File Type Analysis also involves comparing the file extension against the file type from the file signature. If the file extension is for a different file type, the file will be highlighted as suspicious in the analysis report.



An extensive list of file extensions and type signatures can be found at:  
<http://asecuritysite.com/forensics/magic>  
or  
[http://www.garykessler.net/library/file\\_sigs.html](http://www.garykessler.net/library/file_sigs.html)

## 5.1 File Type Analysis

To read the first 4 bytes of a zip file, code such as the following can be used: (create a test zip file called usb.zip to try it out)

```
>>> filename = r'c:\temp\usb.zip'
>>>
>>> fh = open(filename, 'rb')
>>> file_sig = fh.read(4)
>>> file_sig
b'PK\x03\x04'
```

**It is important to read the file in binary mode, because we need the bytes themselves.** (If you forget the 'b', Python will attempt to decode the bytes it is reading, and that will throw an error in this case.)

To compare the 4 byte against the hex signature for a Zip File, code such as the following can be used:

```
>>> ZIPFILE = b'\x50\x4B\x03\x04'
>>> if file_sig == ZIPFILE:
    print('File is a Zip File')
```

The Python **binascii** module allows conversion of ASCII chars to Hex and Binary.

The **.hexlify()** function takes an ASCII string and outputs a string of hex.

```
>>> import binascii
>>>
>>> binascii.hexlify(file_sig)
b'504b0304'
>>> filesig_hex = binascii.hexlify(file_sig)
>>> if filesig_hex == '504b0304':
    print('File is a zip file')
```

```
File is a zip file
>>>
```

**Q:** How many bytes long is the file\_sig string?

**Q:** How many Hex chars represent this?

**Q:** How many Hex chars are used to represent a byte of data from the file?

To quickly view an entire file in Hex: (try this only with really small files and remember if you have read 4 bytes from the same file already you need to reset the read location to 0 first!):

```
>>> binascii.hexlify(fh.read())
b'504b0304140000000800a4635e4bf5576f6124000000300000000b0000006f757466696c652e7478744d
c6b90d00000400c05e62073b7836524844c3fed189ab6ebc8732ca19e12eeffa6e080b504b01023f0014000
0000800a4635e4bf5576f6124000000300000000b002400000000000002000000000000006f757466696
c652e7478740a002000000000000100180047145bae7a51d301079529277a51d301079529277a51d30150
4b050600000000010001005d0000004d0000000000'
```

## 5.2 File Type Analysis Script

From the IDLE Shell, using **File>New Window**, create a script **file\_type.py**, based on the code below:

```
# Script: file_type_sig.py
# Desc:   Check file type signature against filename extension.
# Modified: Nov 2018 (PEP8 compliance)
#
import sys
import os
import binascii

file_sigs = {b'\xFF\xD8\xFF': ('JPEG', 'jpg'), b'\x47\x49\x46': ('GIF', 'gif')}

def check_sig(filename):
    """checks the file type signature of the file passed in as arg,
    returning the type of file and correct extension in a tuple """

    # Read File
    # ... YOUR CODE ... open file, and read 3 bytes
    print('[*] check_sig() File:', filename, end=' ')
    # print('Hash Sig:', binascii.hexlify(file_sig))

    # Check for file type sig
    # ... YOUR CODE ... to check type not in dic

    # File Type Sig found, so get sig and ext from file_sigs dic
    # ... YOUR CODE ... to get tuple from dic

    # Check if Type matches valid file extension
    # ... YOUR CODE ...

    # Valid ext
    # ... YOUR CODE ...
    print('[+] Extension:', os.path.splitext(filename)[1], end=' ')
    # print('    Actual File type:', file_type)

def main():
    # temp testing url argument
    sys.argv.append(r'c:\temp\invalid.txt')

    # Check args
    if len(sys.argv) != 2:
        print('usage: file_sig filename')
        sys.exit(1)

    file_hashsig = check_sig(sys.argv[1])

if __name__ == '__main__':
    main()
```

To test the code, create a word document and save it to c:\temp\invalid.docx. Then run your code with the hard coded test argument of the invalid.docx file. It should print:

```
>>>
[*] check_sig() File: c:\temp\invalid.docx
>>> |
```

**Q:** Did you get the correct output as shown above?

Now add your own code, to

- open the file
- read the first 3 bytes of the file into a file signature variable.
- Uncomment the print statement and it should now print the 3 byte file signature shown below:

```
[*] check_sig() File: c:\temp\invalid.docx Hash Sig: b'504b03'
```

**Q:** Did you get the output with the 3 byte signature as shown above?

Now add code to check if the file signature is not in the dictionary. In that case the code should return a tuple with (-1,"") and print the following:

```
| [*] check_sig() File: c:\temp\invalid.docx Hash Sig: b'504b03'
| [-] File type not identified - file sig not in db
```

Use an if statement, and in it code similar to:

```
# file sig not found
print('[-] File type not identified - file sig not in db')
return (-1,'')
```

Test the code with the same .docx file.

**Q:** Did you get the correct output as shown above?

Now add code for a file in the dictionary, which gets the extension using file signature as the key to the dictionary, and then checks if the type returned does not match the file extension of the test file. If it does not match, return a tuple with (-2,file\_type,file\_ext) and print a message.

Use code similar to:

```
print(f'[+] File type identified as {file_type}')
print(f'[+] Extension: {os.path.splitext(filename)[1]}')
print(f'[!] Expected : {file_ext}. Investigation recommended.')
return (-2,file_type,file_ext)
```

If tested with a jpeg file, which has had the extension changed, it should print output similar to:

```
[*] check_sig() File: c:\temp\cyber.dll Hash Sig: b'ffd8ff'
[+] File type identified as JPEG
[+] Extension: .dll
[!] Expected : .jpg. Investigation recommended.
>>>
```

Test the code, by downloading a small jpeg from google images, saving it to c:\temp\validjpg.jpg, and changing the extension to .dll. Then run the code with F5.

Q: Did you get the correct output as shown above?

Now add code for a file in the dictionary where the extension matches the file extension of the test file. Return a tuple with (0,file\_type,file\_ext) and print a message.

Use code similar to:

```
print(f'[+] File type identified as {file_type}')
print(f'[+] Extension:', os.path.splitext(filename)[1])
print(f'[ ] OK - valid extension for this file type')
return (0,file_type,file_ext)
```

If tested with a valid jpeg file, it should print output similar to:

```
[*] check_sig() File: c:\temp\cyber.jpg Hash Sig: b'ffd8ff'
[+] File type identified as JPEG
[+] Extension: .jpg
[ ] OK - valid extension for this file type
>>>
```

Test the code, by downloading a jpeg from google images, and saving it to c:\temp\valid.jpg, and running code with F5.

Q: Did you get the correct output as shown above?